

# Computing with Hereditarily Finite Sequences

Paul Tarau

Department of Computer Science and Engineering  
University of North Texas  
*tarau@cs.unt.edu*

**Abstract.** We use Prolog as a flexible meta-language to provide executable specifications of some fundamental mathematical objects and their transformations. In the process, isomorphisms are unraveled between natural numbers and combinatorial objects (rooted ordered trees representing hereditarily finite sequences and rooted ordered binary trees representing Gödel’s System **T** types).

This paper focuses on an application that can be seen as an unexpected “paradigm shift”: we provide recursive definitions showing that the resulting representations are directly usable to perform symbolically arbitrary-length integer computations.

Besides the theoretically interesting fact of “breaking the arithmetic/symbolic barrier”, the arithmetic operations performed with symbolic objects like trees or types turn out to be genuinely efficient – we derive implementations with asymptotic performance comparable to ordinary bitstring implementations of arbitrary-length integer arithmetic.

The source code of the paper, organized as a literate Prolog program, is available at <http://logic.cse.unt.edu/tarau/research/2011/pPAR.pl>

**Keywords:** *modeling finite mathematics in logic programming, symbolic arbitrary precision arithmetic, ranking/unranking of hereditarily finite sequences, balanced parenthesis languages,*

## 1 Introduction

This paper exhibits a creative use of logic programming as a modeling tool for several interesting concepts at the intersection of combinatorics, formal languages, foundation of mathematics and coding theory. It builds on the declarative data transformation framework introduced in [1,2], where we introduce a methodology to derive bijective mappings between fundamental data types used in programming languages (sets, multisets, sequences to graphs, digraphs, DAGs, hypergraphs etc.)

At the same time, with practical uses for arbitrary size integer arithmetic in mind, we will focus on keeping the asymptotic complexity of various operations similar to that of operations on conventional bitstrings.

Like [1], this paper is organized as a literate Prolog program. This means that our “lingua franca” is logic programming rather than the usual mathematical notation.

It has been a long tradition in logic programming to model program properties and behaviors in terms of mathematical reasoning. We pay it back this time, and model some intriguing mathematical concepts as logic programs.

The paper is organized as follows.

Section 2 overviews, following [1] a bijection between natural numbers and sequences that is extended in section 3, by recursive application, to hereditarily finite sequences. Section 4 describes a novel way to perform arbitrary length arithmetic computations using multiway tree representations of hereditarily finite sequences and discusses some potential applications for implementation of arithmetic operations with numbers that do not fit in computer memory with conventional binary encodings. It is followed by a sketch of similar mechanism in section 5 for the type language of Gödel's system **T**. Section 6 introduces a bijection between hereditarily finite sequences and balanced parenthesis languages providing a succinct representation for them. Sections 7 and 8 discuss related work and conclude the paper.

## 2 A bijection between finite sequences and natural numbers

Let  $\mathbb{N}$  be the set of natural numbers and  $[\mathbb{N}]$  the set of finite sequences of natural numbers (that can also be seen as the set of functions from an initial segment of  $\mathbb{N}$  to  $\mathbb{N}$  - or even more generally, as *finite functions*). We will first derive, following [1] a bijection  $\mathbb{N} \rightarrow [\mathbb{N}]$ .

We define the following predicates working on natural numbers:

```

cons(X,Y,XY):-X>=0,Y>=0,XY is (1+(Y<<1))<<X.

hd(XY,X):-XY>0,P is XY /\ 1,hd1(P,XY,X).

    hd1(1,_,0).
    hd1(0,XY,X):-Z is XY>>1,hd(Z,H),X is H+1.

tl(XY,Y):-hd(XY,X),Y is XY>>(X+1).

null(0).

```

After observing that the relations  $\text{cons}(X,Y,Z)$ ,  $\text{hd}(Z,X)$ ,  $\text{tl}(Z,Y)$  hold if and only if  $Z = 2^X(2Y + 1)$ , it can be proven by structural induction that:

**Proposition 1** *The predicates  $\text{cons}/3$ ,  $\text{hd}/2$ ,  $\text{tl}/2$ ,  $\text{null}/1$  emulate the list functions  $\text{CONS}$ ,  $\text{CAR}$ ,  $\text{CDR}$ ,  $\text{NIL}$  as defined in [3] (see proof in [1]).*

Using these predicates we define a bijection between finite sequences represented as lists of their values and natural numbers

```

list2nat([],0).
list2nat([X|Xs],N):-list2nat(Xs,N1),cons(X,N1,N).

```

```

nat2list(0, []).
nat2list(N, [X|Xs]):-N>0,hd(N,X),tl(N,T),nat2list(T,Xs).

```

working as follows:

```

?- nat2list(2012,Ns),list2nat(Ns,N).
Ns = [2, 0, 0, 1, 0, 0, 0, 0],
N = 2012

```

### 3 Ranking Hereditarily Finite Sequences

**Definition 1** *The ranking problem for a family of combinatorial objects is finding a unique natural number associated to each object, called its rank. The inverse unranking problem consists of generating a unique combinatorial object associated to each natural number.*

**Definition 2** *A hereditarily finite sequence is [] or a finite sequence of hereditarily finite sequences.*

We will describe, by instantiating the data type transformation described in [1] how to extend a bijection  $\mathbb{N} \rightarrow [\mathbb{N}]$  to trees representing *hereditarily finite sequences*. The two sides of the bijection are expressed as two higher order predicates **rank** and **unrank** parameterized by two transformations F and G:

```

unrank(F,N,Rs):-call(F,N,Ns),maplist(unrank(F),Ns,Rs).

rank(G,Ts,Rs):-maplist(rank(G),Ts,Xs),call(G,Xs,Rs).

```

These predicates can be seen as a form of “structured recursion” that propagate a simpler operation (F and G) guided by the structure of the underlying data type. We can instantiate this mechanism to derive a bijection between natural numbers and trees representing hereditarily finite sequences using **rank** and **unrank** as:

```

nat2hfseq(N,T):-unrank(nat2list,N,T).

hfseq2nat(T,N):-rank(list2nat,T,N).

```

They work as follows:

```

?- nat2hfseq(2012,HFSEQ),hfseq2nat(HFSEQ,N).
HFSEQ = [[[[[]]], [], [], [[]], [], [], [], []],
N = 2012

```

One can represent the recursive *unfolding* of a natural number into a hereditarily finite sequence as a directed ordered multigraph (Fig. 1). Note that as the mapping **nat2list** generates a sequence where the order of the edges matters, this order is indicated with integers starting from 0 labeling the edges.

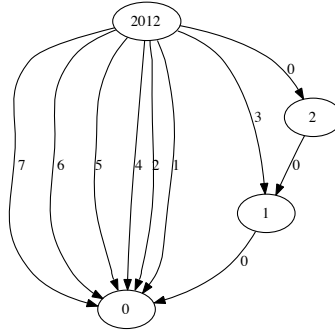


Fig. 1: 2012 as a HFSEQ

## 4 Computing with hereditarily finite sequences

This section describes a surprising possibility derived from the existence of bijections between various data types and natural numbers. It answers positively the following question: can we turn such bijections into actual isomorphisms such that operations like additions or multiplications defined on symbolic objects (e.g. trees or parenthesis languages) mimic their natural number equivalents? Moreover, we want a genuinely constructive proof that this can be done, which means that we need to build inductive definitions, starting with successor and predecessor and then extend them to implement everything else.

We will build these operations incrementally. We start with successor/predecessor operations and simple (but slow) mappings to natural numbers. We then provide efficient implementations, working, like in the case of bitstring representations, in time proportional to the size of the operands.

### 4.1 Successor and predecessor

To derive efficient successor and predecessor operations we recall that the equation  $Z = [X|Y]$  on hereditarily finite sequences corresponds bijectively to the equation

$$Z = 2^X(2Y + 1) \quad (1)$$

on natural numbers. Successor and predecessor predicates  $s/2$  and  $p/2$  are defined as:

```

s([], []).
s([K|Ks] | Xs, [], K1 | Xs) :- p([K|Ks], K1).
s([[] | Xs, [K1|Ks] | Ys) :- s(Xs, [K|Ys]), s(K, [K1|Ks]).

p([], []).
p([[] | K | Xs, [K1|Ks] | Xs) :- s(K, [K1|Ks]).
p([K|Ks] | Xs, [] | Zs) :- p([K|Ks], K1), p([K1|Xs], Zs).

```

The two predicates are deterministic and implement functions when their first arguments are ground, given that the patterns used in the heads of the rules share no

instances. If executed under a breadth-first evaluation rule (or if impure Prolog operations are used) the two predicates can be merged into a single reversible predicate. We have preferred pure Horn clause definitions, however, and reordered the goals in the clause bodies as needed.

When navigating over hereditarily finite sequence trees, **s/2** implements tree transformations such that the following propositions hold:

**Proposition 2** *If  $T$  is such that  $\text{hfseq2nat}(T, N)$ ,  $\text{s}(T, T1)$  and  $\text{hfseq2nat}(T1, N1)$  hold, then  $N1$  is  $N+1$ .*

**Proposition 3** *If  $T$  (assumed different from  $[]$ ) is such that  $\text{hfseq2nat}(T, N)$ ,  $\text{p}(T, T1)$  and  $\text{hfseq2nat}(T1, N1)$  hold, then  $N1$  is  $N-1$ .*

One can rephrase this saying that the pair **hfseq2nat** and **nat2hfseq** acts as a *iso-functor* that transports successor and predecessor operations between natural numbers and hereditarily finite sequences. A proof is obtained by structural induction on the first argument of the two predicates after defining a mapping between a multiway tree type and a natural number type supporting an axiomatization of Peano arithmetic. Note that by replacing  $[]$  by 0 and each relation of the form  $[X|Y]=Z$  in the inductive definition of **s** and **t** with equations of the form  $2^X * (2 * Y + 1) = Z$  one can obtain arithmetic formulas that, after simplifications result in the the usual arithmetic relations defining **s** and **p**.

One can prove the correctness of **s** and **p** with respect to the corresponding successor and predecessor operations on  $\mathbb{N}$ , by verifying that when interpreting each constructor in terms of equation 1 on  $\mathbb{N}$  the resulting formulas become identities.

For instance,  $\text{s}([], [])$  becomes  $s(0, 2^0 * (2 * 0 + 1))$  and then  $s(0, 1)$  which states that the successor of 0 is 1.

On the other hand the second and third recursive equations in the definitions of **s** and **p** become logical implications between arithmetic identities, relatively easy to prove through a sequence of simplifications.

For instance, the second equation in the definition of **s/2** becomes, after putting  $[K|Ks] \rightarrow x, Xs \rightarrow y, K1 \rightarrow z$  with  $x, y, z \in \mathbb{N}$ .

$$s([x|y], [0, z|y]) : \neg p(x, z). \quad (2)$$

After interpreting  $:-$  as inverse logical implication  $\Leftarrow$  we obtain

$$s(2^x * (2 * y + 1), 2^0 * 2 * (2^z * (2 * y + 1)) + 1) \Leftarrow p(x, z). \quad (3)$$

After interpreting **s** and **p** as successor and predecessor on  $\mathbb{N}$  we obtain:

$$1 + (2^x * (2 * y + 1)) = 2 * 2^z * (2 * y + 1) + 1 \Leftarrow (x = z + 1). \quad (4)$$

After replacing  $x$  by  $z + 1$  on the left side we obtain:

$$2^{z+1} * (2 * y + 1) = 2^{z+1} * (2 * y + 1) \quad (5)$$

which is clearly an identity in  $\mathbb{N}$ .

Note that the ability to reason about the correctness of our programs has been clearly facilitated by the declarative semantics of Prolog, for instance when interpreting  $:-$  as reverse logical implication.

Let us now define, using **s/2** and **p/2** a simple (but inefficient) bijection from trees (with leaves made of empty lists) to ordinary natural numbers:

working as follows:

After defining a generator for the infinite stream of hereditarily finite sequences mapped to successive natural numbers

one can confirm empirically that our two symbolic  $\mathbf{s}/2$  and  $\mathbf{p}/2$  operations provide indeed emulations of their standard counterparts:

## 4.2 Simple arithmetic operations in terms of successor and predecessor

It works indeed as expected:

We will next define efficient operations, with asymptotic complexity comparable to typical bignum packages provided by various languages.

We start with recognizers for odd numbers  $\mathbf{o}/2$ , strictly positive even numbers  $\mathbf{i}/2$  and zero  $\mathbf{e}/1$ .

```

o_([_] | _).
i_([_] | _).
e_().

```

Next, we define our constructors. The first one,  $o/2$  builds odd numbers, as if provided by leftshift+increment operation  $2*X+1$ . The later applies the successor predicate to the result of the first, as if provided by the  $2*X+2$  operation.

```

o(X, [_ | X]).
i(X, Y) :- s([_] | X), Y.

```

Note that the predicate  $e_/1$  can also be seen as a constructor for the empty list representing 0.

#### 4.4 Arithmetic operations with hereditarily finite sequences – efficiently

To provide efficient, possibly practical implementations of arithmetic operations, we will need a few more steps towards emulating binary representations including variants of left and right shifting operations.

**Deconstructing** Let us first build a deconstructor  $r/2$ , working as a decrement + rightshift operation on bitstrings such that it maps both  $2*X+1$  and  $2*X+2$  to  $X$ , i.e. such that it reverses the action of the constructors  $o/2$  and  $i/2$ .

```

r([_] | Xs), Xs.
r([X | Xs] | Ys), Rs :- p([X | Xs] | Ys), [_ | Rs].

```

Note that the first clause maps to  $n$  a term corresponding to an odd number of the form  $2*n+1$ , while the second applies the predecessor to an even number while trimming the result (an odd number) in a similar way to the first clause.

**Converting back and forth** Given the deconstructor  $r/2$  and the constructors  $o/2$  and  $i/2$ , we can empirically validate the intuitions behind our symbolic representations, by mapping them one-to-one to conventional natural numbers.

We first define a converter  $s2n/2$ , mapping tree representations of hereditarily finite sequences to conventional natural numbers:

```

s2n([], 0).
s2n(X, R) :- o_(X), r(X, S), s2n(S, N), R is 1+2*N.
s2n(X, R) :- i_(X), r(X, S), s2n(S, N), R is 2+2*N.

```

then a converter  $n2s/2$  from natural numbers to our symbolic representations:

```

n2s(0, []).
n2s(N, R) :- N > 0, P is N mod 2, N1 is (N-1) // 2,
    n2s(N1, X),
    ( P = 0 => i(X, R)
    ; o(X, R)
    ).

```

They work as expected, and `s2n` can be seen as enumerating the stream of natural numbers correctly.

```
?- n2s(42,S),s2n(S,N).
S = [[[]], [[]], [[]]],
N = 42
```

```

?-n(X),s2n(X,N).
X = [], N = 0 ;
X = [[]], N = 1 ;
X = [[[]]], N = 2 ;
X = [[], []], N = 3 ;
.....

```

Note also that they work in time proportional to the size of the representations.

**Efficient Addition** Guided by this mapping, that sees our symbolic representations as if they were bitstrings in *bijective base-2*, we can implement an addition operation working in time proportional to the size of the operands:

```

a([],Y,Y).
a([X|Xs],[],[X|Xs]).
a(X,Y,Z):-o_-(X),o_-(Y),a1(X,Y,R), i(R,Z).
a(X,Y,Z):-o_-(X),i_-(Y),a1(X,Y,R), a2(R,Z).
a(X,Y,Z):-i_-(X),o_-(Y),a1(X,Y,R), a2(R,Z).
a(X,Y,Z):-i_-(X),i_-(Y),a1(X,Y,R), s(R,S),i(S,Z).

a1(X,Y,R):-r(X,RX),r(Y,RY),a(RX,RY,R).
a2(R,Z):-s(R,S),o(S,Z).

```

working instantly on arbitrarily large natural numbers:

```
?-n2s(12345678901234567890,A),n2s(100000000000000000000,B),a(A,B,S),s2n(S,N).
A = [[[]], [[]]], [[]], [], [[]], [[]], [[...]], [], []|...],
B = [[[], [], [[]]], [[]], [], [], [], [[]], [[]], [[]]...|...],
S = [[[]], [[]]], [[]], [], [[]], [[]], [[...]], [], []|...],
N = 22345678901234567890 .
```

**Efficient Multiplication** We can implement efficient multiplication guided by intuitions about binary multiplication in base 2 and bijective-base 2 as follows:

```

m([],_,[]).
m(_,[],[]).
m(X,Y,Z):-p(X,X1),p(Y,Y1),m0(X1,Y1,Z1),s(Z1,Z).

m0([],Y,Y).
m0([[]|X],Y,[[]|Z]):-m0(X,Y,Z).
m0(X,Y,Z):-i(X),r(X,X1),m0(X1,Y,Z1),a(Y,[[]|Z1],Y1),s(Y1,Z).

```

One can see that it handles easily large numbers (the *googol*= $10^{100}$  included!):





```

p_((e→e), e).
p_((e→(K→Xs)), ((K1→Ks)→Xs)) :- s_(K, (K1→Ks)).
p_(((K→Ks)→Xs), (e→Zs)) :- p_((K→Ks), K1), p_((K1→Xs), Zs).

```

The following example illustrates that `s_` and `p_` work as expected:

```

?- s_(e,One),s_(One,Two),s_(Two,Three),s_(Three,Four),p_(Four,Three).
One = (e→e),
Two = ((e→e)→e),
Three = (e→e→e),
Four = (((e→e)→e)→e)

```

We will only give here the code of a generator `n_/1` for the infinite stream of natural numbers represented as types in system **T**, and a simple converter to usual natural numbers `t2n`, modeled after `tree2nat/2`.

```

n_(e).
n_(S):-n_(P),s_(P,S).

t2n(e,0).
t2n((T→S),N):-p_((T→S),U),t2n(U,M),N is M+1.

```

confirming empirically that our computations mimic the usual ones:

```

?- n_(T),t2n(T,N).
T = e, N = 0 ;
T = (e→e), N = 1 ;
T = ((e→e)→e), N = 2 ;
T = (e→e→e), N = 3 ;
T = (((e→e)→e)→e), N = 4 ;
...

```

Fast arithmetic computations, operating directly on types, can be derived using the corresponding code for hereditarily finite sequences as “boilerplate”.

*Deriving a bidirectional successor/predecessor predicate* The predicates `s_` and `p_` are mutually recursive and structurally similar. Moreover, each of them would run reversibly under a breadth-first evaluation order. An interesting challenge is to derive a bidirectional variant replacing both predicates. One could achieve this by using impure operations like `nonvar/1` to check which argument is instantiated or, equivalently, checking the instantiation of the arguments using negation as failure. We proceed by merging the two predicates’ shared clauses and adding an extra argument taking the values `up` or `down` to indicate which way the computation goes.

```

sp(e, (e→e), _).
sp(((K→Ks)→Xs), (e→(K1→Xs)),Dir) :-
    flip(Dir,Other),
    sp(K1,(K→Ks), Other).
sp((e→Xs), ((K1→Ks)→Ys), up) :-
    sp(Xs, (K→Ys),up),
    sp(K, (K1→Ks), up).

```

```

sp((e→Xs), ((K1→Ks)→Ys), down) :-
    sp(K, (K1→Ks), down),
    sp(Xs, (K→Ys), down).

```

```

flip(up,down).
flip(down,up).

up_or_down(_X,Y,down):- \+(Y=other).
up_or_down(X,_Y,up):- \+(X=other).

sp(X,Y):-up_or_down(X,Y,Dir),sp(X,Y,Dir).

```

Note also the auxiliary predicate `flip/2`, which indicates a change of direction, and the auxiliary predicate `up_or_down`, that chooses among the two possible directions, based on the instantiation of at least one of the arguments of `sp/2`. We detect instantiation of the arguments testing them against the atom `other`, assumed not to be part of the Herbrand Universe of our program.

One step further, we push the call to `sp/3` into `flip/2` (as it is the only continuation of `flip/2`), and merge the last two clauses, while delegating the ordering of the recursive calls to the auxiliary predicate `order_sp`. Note that we also fold `up_or_down` as part of the definition of `sp/2`.

```

sp(e, (e→e), _).
sp(((K→Ks)→Xs), (e→(K1→Xs)), Dir):-flip_sp(Dir, K1, (K→Ks)).
sp((e→Xs), ((K1→Ks)→Ys), Dir):-order_sp(Dir, Xs, (K→Ys), K, (K1→Ks)).

flip_sp(up,X,Y) :- sp(X,Y,down).
flip_sp(down,X,Y) :- sp(X,Y,up).

order_sp(up,A,B,C,D) :- sp(A,B,up), sp(C,D,up).
order_sp(down,A,B,C,D) :- sp(C,D,down), sp(A,B,down).

sp(X,Y) :- \+(X=other), sp(X,Y,up).
sp(X,Y) :- \+(Y=other), sp(X,Y,down).

```

One can try out `sp/2` working as a bidirectional successor/predecessor predicate when at least one of its arguments is instantiated:

```

?- sp(Pred,((e→e)→e)).
Pred = (e→e) .

```

```

?- sp((e→e),Succ).
Succ = ((e→e)→e) .

```

```

?- sp((e→e),((e → e) → e)).
true.

```

## 6 Mapping hereditarily finite sequences to parenthesis languages

We will next explore the bijection between hereditarily finite sequences and the language of balanced parenthesis, known to combinatorialists [5,6,7] as a member of the *Catalan family*, which also includes the binary trees representing **System T** types.

An encoder for the balanced parenthesis language is obtained by combining a parser and a writer, which, with some ingenuity, can be made one and the same in a language like Prolog.

As hereditarily finite sequences naturally map one-to-one to parenthesis expressions expressed as bitstrings, we will choose them as target of the transformers. Our parser recurses over a bitstring (encoding balanced parentheses ' [' as 0, ']' as 1) and builds a HFSEQ tree T:

```

pars_hfseq(Xs,T):-pars2term(0,1,T,Xs,[]).

pars2term(L,R,Xs)  -> [L],pars2args(L,R,Xs).

pars2args(_,R,[]) -> [R].
pars2args(L,R,[X|Xs])>pars2term(L,R,X),pars2args(L,R,Xs).

```

Note that `pars_hfseq` is *bidirectional* i.e. it works both as an encoder and a decoder:

```

?- pars_hfseq([0,0,1,0,1,1],T),pars_hfseq(Ps,T).
T = [[], []],
Ps = [0, 0, 1, 0, 1, 1]

```

One can see the bijection defined by `pars_hfseq` as a bridge between a family of formal languages and hereditarily finite sequences, represented as multiway trees.

*Kraft's inequality* As the sequences computed by `pars_hfseq` are elements of the balanced parenthesis language (also called Dyck primes) [8], they implement *uniquely decodable self-delimiting* codes. Moreover, each of them is also a *prefix code*, i.e. there's no way to add a string made of any combination of balanced left or right parenthesis at the end of a code and obtain another code. For a similar reason, each of them is also a *suffix code*. Such codes are known in the literature under a variety of different names i.e. as *reversible variable-length codes*, *bifix codes* or *fix-free codes*<sup>2</sup>.

In particular, given that they are *uniquely decodable* codes, it follows that the *Kraft inequality* [9] holds for them, i.e. if  $l_0, l_1 \dots l_k \dots$  denote the length of the codes, then

$$\sum_{k \geq 0} 2^{-l_k} \leq 1 \quad (6)$$

We define the function computing the left side of the *Kraft inequality* (called *Kraft-sum*), and the corresponding test as follows.

<sup>2</sup> A nice property of such codes is that parallel bidirectional decoding is possible. Also, the ability to decode from either the beginning or the end makes them suitable for encoding media streams.

```

kraft_sum(M,S):- M1 is M-1, numlist(0,M1,Ns),
    maplist(kraft_term,Ns,Ls),
    sumlist(Ls,S).

kraft_term(N,X):-parsize(N,L), X is 1/2^L.

parsize(N,L):- nat2hfseq(N,HFSEQ), pars_hfseq(Xs,HFSEQ), length(Xs,L).

kraft_inequality(M):-kraft_sum(M,S),S<=1.

```

The following example illustrates that the Kraft's inequality holds and it is likely that the Kraft-sum converges to a value below 0.5:

```

?- maplist(kraft_sum,[10,100,1000,2000,3000,4000],R).
R = [0.364258, 0.382935, 0.390383, 0.391615, 0.392292, 0.392598]

```

The bijection between hereditarily finite sequences and balanced parenthesis languages provides a succinct alternative representation for purposes of efficient arithmetic operations using bitvector operations – by encoding the two parenthesis as 0 and 1. As a possible practical application, this allows building in Prolog, at source level, a library supporting arbitrary length arithmetic operations.

## 7 Related work

*Ranking* functions can be traced back to Gödel numberings [10,11] associated to formulae. Together with their inverse *unranking* functions they are also used in combinatorial generation algorithms [12,13]. Natural number encodings of hereditarily finite sets have triggered the interest of researchers in fields ranging from Axiomatic Set Theory and Foundations of Logic to Complexity Theory and Combinatorics [14,15,16].

The encodings of hereditarily finite sets and sequences described in this paper originate in [1,17,18,19]. The key difference is that while in our previous work we use pairs of bijections encapsulated as higher order predicates/functions to define various isomorphisms directly, here we provide actual algorithms for arithmetic operations, ordering etc. while in our previous work the existence of such algorithms was only implied “non-constructively”.

An emulation of Peano and conventional binary arithmetic operations in Prolog, is described in [20]. Their approach is similar as far as a symbolic representation is used. The key difference with this paper is that our operations work on tree structures, and as such, they are not based on previously known algorithms. Our tree-based algorithms are also likely to support parallel execution in a way similar to the powerlists of [21]. Arithmetic computations with types expressed as C++ templates are described in [22] and in online articles by Oleg Kiselyov using Haskell's type inference mechanism. However, the mechanism advocated there is basically the same as [20], focusing on Peano and binary arithmetics. The connection between hereditarily finite sequences and balanced parenthesis languages places them the context of the well known to combinatorialists *Catalan families* [5,6,7].

## 8 Conclusion

We have derived a few algorithms expressing arithmetic computations symbolically, in terms of hereditarily finite sequences and types in Gödel's system **T**.

This has been made possible by extending the techniques introduced in [1] that allow observing the internal working of intricate mathematical concepts through isomorphisms transporting operations between fundamental data types.

At the same time, we have shown that logic programming provides a flexible framework for modeling mathematical concepts from fields as diverse as combinatorics, formal languages, type theory and coding theory.

Arithmetic operations with hereditarily finite sequences are likely to be interesting for hardware (FPGA) implementations of large integer operations used in cryptography. They are also subject to parallelization [21] and can provide computations with giant numbers that do not fit in any computer memory with a flat bitstring representation<sup>3</sup>.

Reversible variable length (bifix) codes like the ones we derived in section 6 have found uses in image and video coding [23] (including MPEG4!). Prefix codes are used in defining modern versions of Kolmogorov complexity [24]. The fact that this property holds, recursively, for arbitrary parts of the code, combined with their *ability to express programming language constructs*, as shown in [1], makes them an interesting alternative to the Elias codes [25] typically used in the field.

## Acknowledgment

We thank NSF (research grant 1018172) for support.

## References

1. Tarau, P.: An Embedded Declarative Data Transformation Language. In: Proceedings of 11th International ACM SIGPLAN Symposium PPDP 2009, Coimbra, Portugal, ACM (September 2009) 171–182
2. Tarau, P.: “Everything Is Everything” Revisited: Shapeshifting Data Types with Isomorphisms and Hylomorphisms. *Complex Systems* (18) (2010)
3. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM* **3**(4) (1960) 184–195
4. Gödel, K.: Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica* **12**(280-287) (1958) 12
5. Berstel, J., Boasson, L.: Formal properties of XML grammars and languages. *Acta Informatica* **38**(9) (2002) 649–671
6. Liebehenschel, J.: Ranking and unranking of a generalized Dyck language and the application to the generation of random trees. *Séminaire Lotharingien de Combinatoire* **43** (2000) 19
7. Bertoni, A., Choffrut, C., Palano, B.: Context-free grammars and XML languages. *Lecture Notes in Computer Science* **4036** (2006) 108
8. Berstel, J., Boasson, L.: Balanced grammars and their languages. *Lecture Notes In Computer Science* (2002) 3–25

---

<sup>3</sup> Something as simple as `[[[[[[[[[]]]]]]]]]` expresses a very large number - as such numbers correspond to towers of exponents of the form  $2^{2^{2^{\dots}}}$ .

9. Kraft, L.: A device for quantizing, grouping, and coding amplitude-modulated pulses. Master's thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering (1949)
10. Gödel, K.: Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. Monatshefte für Mathematik und Physik **38** (1931) 173–198
11. Hartmanis, J., Baker, T.P.: On Simple Goedel Numberings and Translations. In Loeckx, J., ed.: ICALP. Volume 14 of Lecture Notes in Computer Science., Springer (1974) 301–316
12. Martinez, C., Molinero, X.: Generic algorithms for the generation of combinatorial objects. In Rován, B., Vojtas, P., eds.: MFCS. Volume 2747 of Lecture Notes in Computer Science., Springer (2003) 572–581
13. Knuth, D.E.: The Art of Computer Programming, Volume 4, Fascicle 1: Bit-wise Tricks & Techniques; Binary Decision Diagrams. Addison-Wesley Professional (2009)
14. Takahashi, M.o.: A Foundation of Finite Mathematics. Publ. Res. Inst. Math. Sci. **12**(3) (1976) 577–708
15. Kaye, R., Wong, T.L.: On Interpretations of Arithmetic and Set Theory. Notre Dame J. Formal Logic Volume **48**(4) (2007) 497–510
16. Kirby, L.: Addition and multiplication of sets. Math. Log. Q. **53**(1) (2007) 52–65
17. Tarau, P.: A Groupoid of Isomorphic Data Transformations. In Carette, J., Dixon, L., Coen, C.S., Watt, S.M., eds.: Intelligent Computer Mathematics, 16th Symposium, Calculemus 2009, 8th International Conference MKM 2009 , Grand Bend, Canada, Springer, LNAI 5625 (July 2009) 170–185
18. Tarau, P.: Isomorphisms, Hylomorphisms and Hereditarily Finite Data Types in Haskell. In: Proceedings of ACM SAC'09, Honolulu, Hawaii, ACM (March 2009) 1898–1903
19. Tarau, P.: Declarative Combinatorics: Isomorphisms, Hylomorphisms and Hereditarily Finite Data Types in Haskell (January 2009) Unpublished draft, <http://arXiv.org/abs/0808.2953>, updated version at <http://logic.cse.unt.edu/tarau/research/2010/ISO.pdf>, 150 pages.
20. Kiselyov, O., Byrd, W.E., Friedman, D.P., Shan, C.c.: Pure, declarative, and constructive arithmetic relations (declarative pearl). In: FLOPS. (2008) 64–80
21. Misra, J.: Powerlist: a structure for parallel recursion. ACM Transactions on Programming Languages and Systems **16** (1994) 1737–1767
22. Kiselyov, O.: Type arithmetics: Computation based on the theory of types. CoRR **cs.CL/0104010** (2001)
23. Wen, J., Villasenor, J.: Reversible variable length codes for efficient and robust image and video coding. In: Proceedings Data Compression Conference. (1998) 471–480
24. Li, M., Vitányi, P.: An introduction to Kolmogorov complexity and its applications. Springer-Verlag New York, Inc., New York, NY, USA (1993)
25. Elias, P.: Universal codeword sets and representations of the integers. IEEE Transactions on Information Theory **21**(2) (1975) 194–203